UNIVERSITÀ DI PISA

Relazione progetto per il corso di

Sistemi Operativi e Laboratorio

Corso di Laurea in Informatica

A.A 2020/2021

Francesco Botrugno
matr.575413

1. Introduzione e scelte progettuali

Il progetto consiste nella realizzazione di un File Storage dove ogni client può inviare e ricevere file di qualsiasi tipo.

Non c'è un database per gli utenti, i client sono identificati solo dal numero del descrittore di file associato loro automaticamente al momento della connessione; quindi, il primo client connesso sarà sempre, per esempio, il numero 9 e le connessioni contemporanee successive saranno il 10, 11 ecc.
Se il client 9 si disconnette il prossimo client che si connetterà prenderà il suo posto acquisendo id pari a 9.

I client possono scrivere verso il server qualsiasi file sia presente nel loro dispositivo locale utilizzando un percorso assoluto o relativo, al momento del salvataggio del file nel File Storage verrà estratto solo il nome finale del file, quindi ad esempio `./test/test1/send_files/img1.jpg` sarà identificato all'interno dello Storage solo da `img1.jpg` e tutti i client potranno operare su quel file conoscendone soltanto l'ultima parte del percorso.
Alcune operazioni richiedono che il file sia effettivamente esistente sul dispositivo del client, come ad esempio la `writeFile`, che deve prelevare il contenuto per poterlo inviare al server, altre operazioni invece, come la `lockFile`, hanno successo anche se il file non esiste in locale.
Le cartelle dove il client chiede di memorizzare i file letti o espulsi devono essere già presenti.

Un client che si disconnette non chiude né sblocca i propri file, per farlo deve utilizzare esplicitamente le operazioni corrispondenti (`closeFile` e `unlockFile`).

Un file bloccato (locked) può essere espulso durante l'esecuzione dell'algoritmo di rimpiazzamento, questo avviene per impedire ad un singolo client di riempire il File Storage con solo propri file in modalità locked e di bloccare, a quel punto, tutto lo storage.
Un file bloccato non può essere letto neanche dall'operazione `readNFiles`.
L'operazione `readNFiles` ha successo anche se i file non sono stati aperti dal client che esegue l'operazione, in generale qualsiasi altra operazione richiede prima che il file sia stato aperto.

I file aperti con il flag di creazione (`O_CREATE`) non valgono al fine del conteggio del numero di file nel server e non vengono espulsi, poiché sono vuoti e hanno peso pari a zero, il file viene conteggiato solo al momento della prima scrittura.

L'operazione `openFile` con il flag di lock `O_LOCK` fallisce immediatamente se la lock non può essere acquisita e non attende come invece fa la `lockFile`.

La conversione da Byte a Megabyte e viceversa viene fatta moltiplicando o dividendo per 2^{20}

Il processo server termina nel caso di errori fatali (fallimento di lock/unlock, `malloc` ecc.) che vengono fatti galleggiare fino al worker thread che scrive l'esito sul file di log.

2. Il server

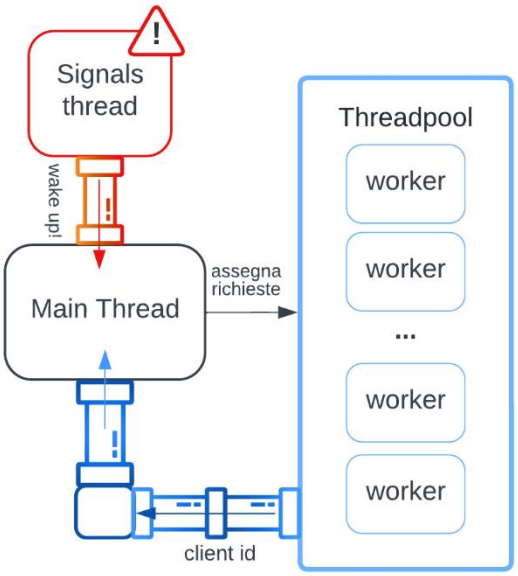
La logica principale del server è contenuta nel file `/src/server.c`, per essere avviato richiede come argomento `-f <file di configurazione>`, il file di configurazione è un file `.txt` e deve avere la seguente struttura:

```
SOCKET_NAME=<percorso socket>
N_WORKERS=<numero workers>
MB_CAPACITY=<capacità in MB>
FILE_CAPACITY=<numero max file>
REPLACE_MODE=FIFO
LOG_FILE=<percorso file di log>
```

Non ci devono essere spazi e i valori numerici devono essere interi. È stata implementata solo la politica FIFO, inserire modalità diverse non altera il comportamento.

La struttura del File Storage si trova nel file `/includes/storage.h`, le strutture dati più rilevanti sono la tabella hash `files` che ha come chiave il nome del file e come contenuto un file di tipo `file_t`, la coda `filenames_queue` che mantiene l'ordine dei file per effettuare l'espulsione della vittima e l'array `clients_waiting_lock` per l'implementazione della `lockFile` spiegata più avanti.

2.1 Threads e segnali



Una volta eseguita la configurazione dei parametri il server fa partire un thread dedicato alla ricezione dei segnali e un threadpool che si occupa di gestire le richieste dei client. Ad ogni thread worker viene passato un oggetto di tipo "`request_t`"(vedere `/includes/request.h`) che contiene tutti i dati necessari per portare a termine la richiesta del client e verrà eseguito il ciclo di operazioni all'interno di "`request_handler.c`".

Il server esegue ininterrottamente la `select` finché non arriva la comunicazione di un segnale o finché non si verifica un errore fatale nel processo.

Tutti i thread comunicano con il main thread attraverso delle pipe, inviando l'id dei client da analizzare nuovamente nella `select` o, nel caso del thread dedicato ai segnali, inviando un messaggio di "wake up!" per dire al server di controllare le variabili `NO_MORE_REQUESTS` e `NO_MORE_CONNECTIONS` e terminare.

2.2 Gestione della concorrenza

Per eseguire correttamente le operazioni sul file storage è stata implementata una semplice lock lettori e scrittori (vedere `/src/read_write_lock.c`) che permette così di effettuare operazioni diverse in contemporanea, l'implementazione può essere migliorata per essere resa più fair.
Le lock sullo storage vengono rilasciate quando possibile, ad esempio se un client chiede di fare la `lockFile` la sequenza delle azioni è la seguente:

> 1) lock in lettura sullo storage → 2) lock in scrittura sul file → 3) rilascia la lock sullo storage → → 4) opera sul file→ 5) rilascia lock sul file

In questo modo dopo l'operazione 3 altri threads possono accedere allo storage senza però poter operare su quello specifico file.
Tutte le funzioni del file `/src/storage.c` sfruttano questo meccanismo.

Per scrivere sul file di log è stata usata una semplice `phtread_mutex_t` che viene acquisita e rilasciata prima e dopo ogni scrittura.

2.3 Formato del file di log

Il file di log è un `.txt`, ogni riga stampata sul file di log ha il seguente formato:

```
/THREAD/=<id thread> /OP/=<operazione> /IDCLIENT/=<id client>
/DELETED_BYTES/=<bytes rimossi> /ADDED_BYTES/=<bytes aggiunti>
/SENT_BYTES/=<bytes inviati al client> /OBJECT_FILE/=<file coinvolto>
/OUTCOME/=<OK> o <ERR>
```

Sono stati usati "/" per impedire errori nel parsing del file: se un file inviato nello storage si fosse chiamato "`OP=READ`" avrebbe falsato i dati sul numero di read effettuate, introducendo "/" come separatore questo non avviene poiché su linux non è consentito inserire lo slash nel nome di un file.

2.4 Le operazioni lockFile e unlockFile

Come da specifica si richiede che le operazioni di lock lascino il client sospeso finché il proprietario non rilascia la lock sul file.

Per implementare questa funzionalità sono state analizzate soluzioni diverse, come l'utilizzo di variabili di condizione con `wait` e `signal`, una `sleep(n)` lato client per poi riprovare a inviare di nuovo la richiesta, l'uso di un thread di supporto aggiuntivo o reinviare immediatamente la richiesta. Ma per ognuna di queste sono stati riscontrati troppi lati negativi.

La soluzione utilizzata è stata quella di aggiungere allo storage un array di dimensione pari al numero massimo di connessioni supportate `MAX_CONN` (si può impostare dal file `/includes/util.h`), ogni volta che una lock fallisce a causa di un errore di tipo `EPERM` l'id del client viene aggiunto all'array e non viene più analizzato dalla `select()` del main thread.

Quando una `unlockFile` termina con successo vengono prelevati, uno per volta, i client all'interno dell'array e vengono riprovate tutte le richieste di lock.

Lato client l'utente invia i dati per l'operazione di lock e aspetta l'esito, tutte le volte che l'errore restituito è `EPERM` rispedirà i dati necessari e attenderà una nuova risposta.

3. Il client

La struttura del client è un processo single-threaded che legge i comandi inseriti da linea di comando ed effettua le operazioni associate, un comando può inviare più richieste verso il server, per esempio il comando -c <nome file> per cancellare un file, prima chiama la `lockFile` e poi la `removeFile`.

Per testare l'operazione di `appendToFile` si è scelto di effettuarla nel caso di fallimento di una `writeFile` dato che non c'era nessun comando esplicito.

Il client aspetta gli esiti restituiti dall'API ed effettua le operazioni di conseguenza. Il client non è in grado di distinguere errori fatali del server da semplici errori di fallimento, continuerà la sua esecuzione finché sarà in grado di comunicare con il server.

4. Tipi di errori

Per gestire gli errori dello storage sono stati scelti i valori di default di errno più appropriati:

`ENOTCONN`: assenza di connessione verso il server
`EISCONN`: connessione già presente
`EINVAL`: argomenti non validi
`ENOENT`: il file o la cartella non è presente (sul server o sul client)
`EEXIST`: il file esiste già sul server
`EACCESS`: accesso vietato perché il file non è stato aperto
`EPERM`: accesso vietato perché il file è bloccato da un altro utente
`ENODATA`: il file non contiene dati
`EALREADY`: l'operazione è già stata effettuata (per esempio una `writeFile` eseguita due volte)
`EBADRQC`: codice richiesta non valido
`EUSERS`: troppi utenti (usata nel meccanismo di lock se l'array è pieno)

5. Info sui test

test 1: ci si aspetta che alcune operazioni falliscano per mancanza di diritti di accesso, come nel caso di un client con id diverso dal detentore della lock.

test 2: dopo l'esecuzione del test nella cartella `test/test2/victims1` devono trovarsi i file `text1.txt`, `text2.txt, text3.txt, text4.txt` e la cartella `test/test2/victims2` non deve essere vuota.

test 3: nel sunto delle statistiche dovrà comparire un numero piuttosto alto di rimpiazzamenti, una memoria massima inferiore a 32 MB e un numero di file circa pari a 23.

6. Repository Github

Link al repository pubblico: https://github.com/FranBot97/progettoSOL